CIS 1100

Objects (Using and starting to make them)!

Python
Spring 2025
University of Pennsylvania

Reminder: Exam!

- Monday, March 3 in class
- Please arrive a bit early if you can—no more than 15 minutes though
- Exam review session on Sunday March 2 from 2:30-4:30 in Towne 100

Review: What is an object/class?

A class in Python is a construct that allows us to "bundle data and functionality together." *

- A class defines a new data type!
- Allows instances of that class to be created.

* *From the Python documentation on classes*

A class consists of:

- Some attributes (also called fields) that store data
- Some functions that operate with these fields

  These allow us to create abstractions that are easier to wrap your head around.

Review: Class as a tool for abstraction

Some features of a class could also be achieved from a tuple,

but consider... Which of these better communicates its purpose?

```
c = (0.5, 0.5, 0.25)
```

Review: Class as a tool for abstraction

Some features of a class could also be achieved from a tuple,

but consider... Which of these better communicates its purpose?

```
c = (0.5, 0.5, 0.25)

c = Circle(x_center = 0.5, y_center = 0.5, radius = 0.25)
```

Review: Attributes

To build a class, we need to decide which attribute we will include in our abstraction.

Let's say we wanted to make an object that represented a Penn course, what attributes might we want to store in that class? What types would they be? **(L11)**

Review: Syntax

If we have an object that we want to access the fields of, we can do so using the `.` operator

```
good_movie = Movie("Actual People", 2021, 84, "Drama", "Digital", 3.2)
the_name = good_movie.name
print(good_movie.name)
if (good_movie.length > 120):
    print("TOO LONG")
good_movie.genre = "twentysomething " + travis.genre
```

(NOTE: we do not use `()` when accessing attributes directly. `()` is usually used to indicate some sort of function call)

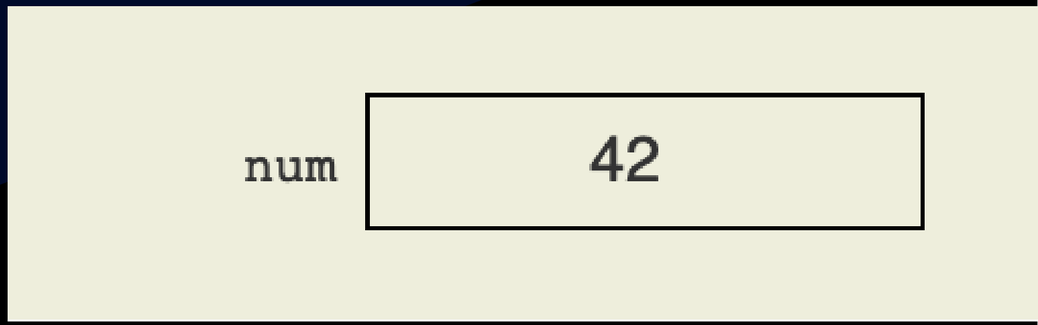
Practice:

Wich of these are (A) *method* calls, (B) accessing *attributes*, or (C) neither

- (M1) `name.upper()`
- (M2) `my_movie.name`
- (M3) `my_move.price_adjust_inflation(2020)`
- (M4) `penndraw.set_pen_color(penndraw.BLACK)`
- (M5) `len(name)`
- (M6) `number.numerator`

Variables, Before

A variable is like a "box" inside of which a piece of data is placed.



Variables, Now

A variable is a **named portion of memory** that contains data of a particular type.

Variables do not directly contain data. Instead, data is stored in a separate portion of the computer's memory.

Instead of storing the data directly, variables of these types tell us how to find the data elsewhere!

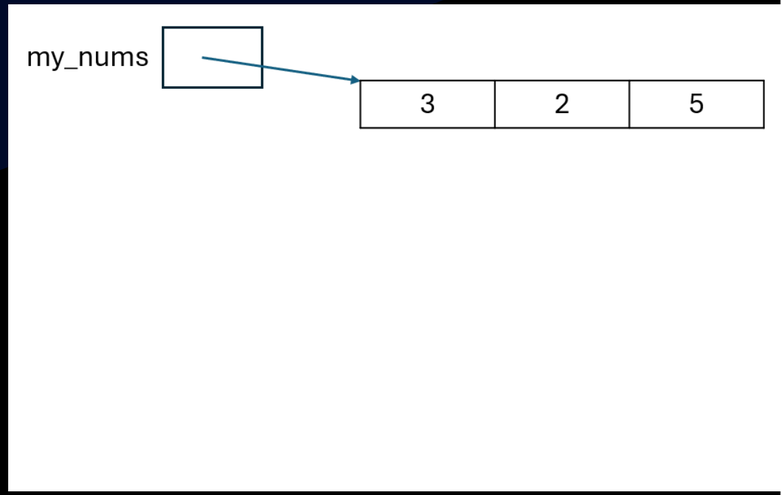
Let's drill down.

All Types Are Reference Types

References

- Reference variables **do not store simple values directly!**
- Reference variables store a **reference** to some object
  - Literally: an address that describes where the object is stored in the computer's memory.
- The object that the reference refers to is known as its ***pointee***

```
my_nums = [3]
my_nums.append(2)
my_nums.append(5)
```



Mutability

Some types are designed to be immutable types. `string`, `int`, `float`, `bool`, `tuple`*.

Even if we pass a reference to them, we cannot modify them.

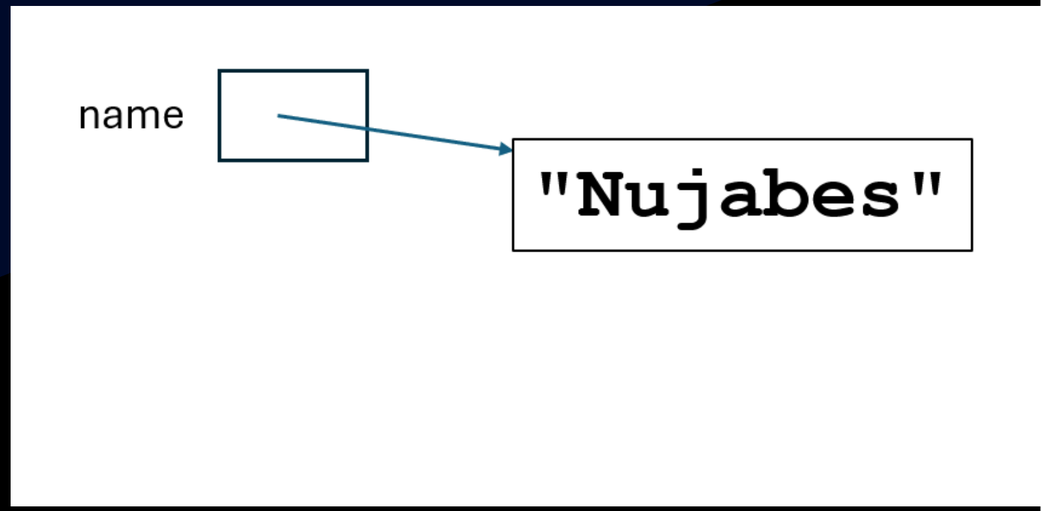
```
number = 5
x = number + 3   # number is not changed, it's value is used as part of a computation
number += 2      # equivalent to number = number + 2, similar to previous line


name = "Nujabes"
name.upper()          # does nothing, returns a new string "NUJABES"
name = name.upper()   # Reassigns name to a new string
```

Memory Diagram: Immutable Type

Lets look at the string a little closer

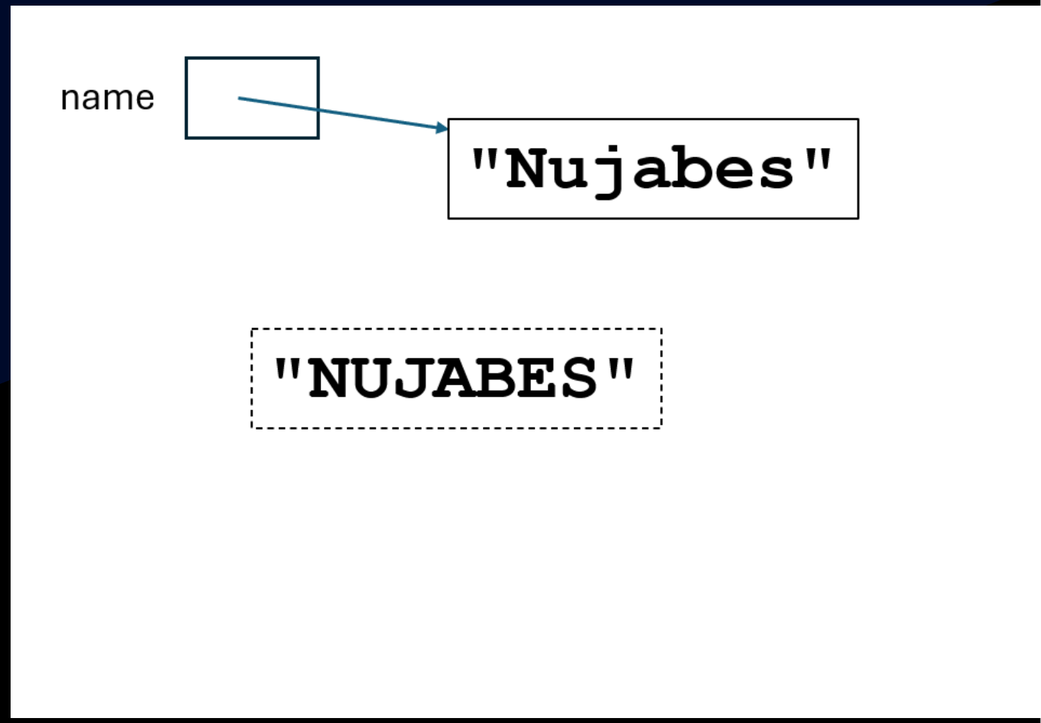
```
name = "Nujabes"     <-
name.upper()              # does nothing, returns a new string "NUJABES"
name = name.upper()       # Reassigns name to a new string
```



Memory Diagram: Immutable Type

Lets look at the string a little closer

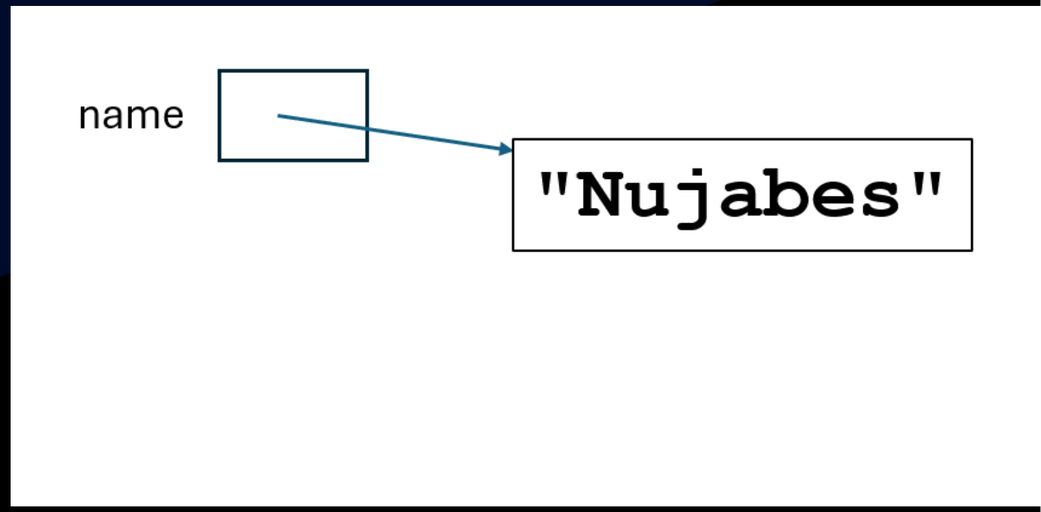
```
name = "Nujabes"
name.upper()        <-  # does nothing, returns a new string "NUJABES"
name = name.upper()     # Reassigns name to a new string
```



Memory Diagram: Immutable Type

Lets look at the string a little closer

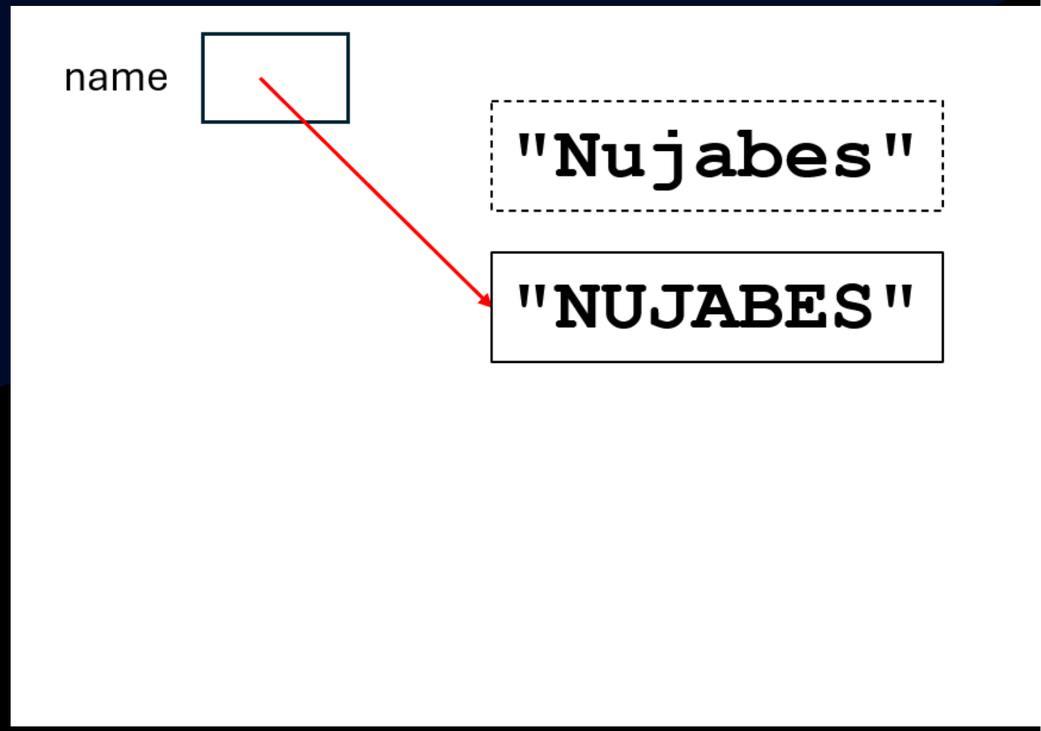
```
name = "Nujabes"
name.upper()          <-  # does nothing, returns a new string "NUJABES"
name = name.upper()       # Reassigns name to a new string
```



Memory Diagram: Immutable Type

Lets look at the string a little closer

```
name = "Nujabes"
name.upper()            # does nothing, returns a new string "NUJABES"
name = name.upper()  <- # Reassigns name to a new string
```



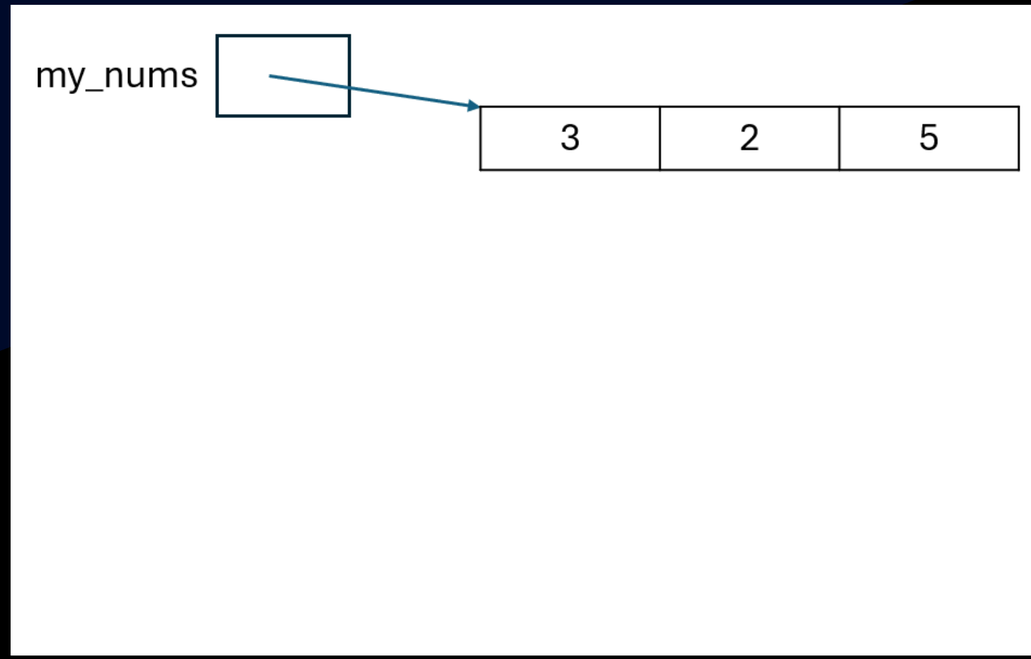
References to Mutable Types

References get more tricky when we start thinking about mutable types.

Consider:

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]      # <-------
    other = my_nums
    func(my_nums)
    other[1] = 1100
    print(my_nums)
```



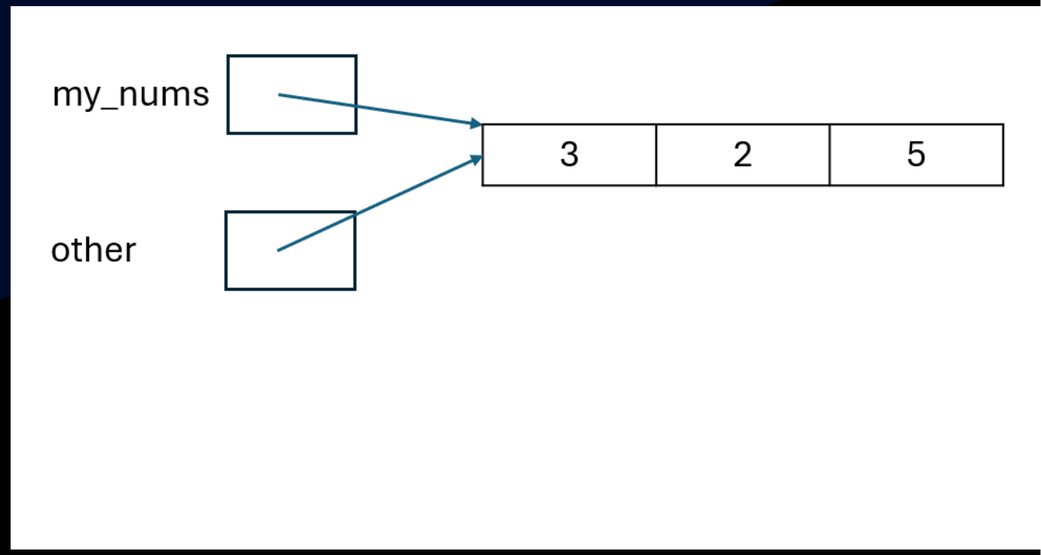
References to Mutable Types

References get more tricky when we start thinking about mutable types.

Consider:

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums          # <-------
    func(my_nums)
    other[1] = 1100
    print(my_nums)
```



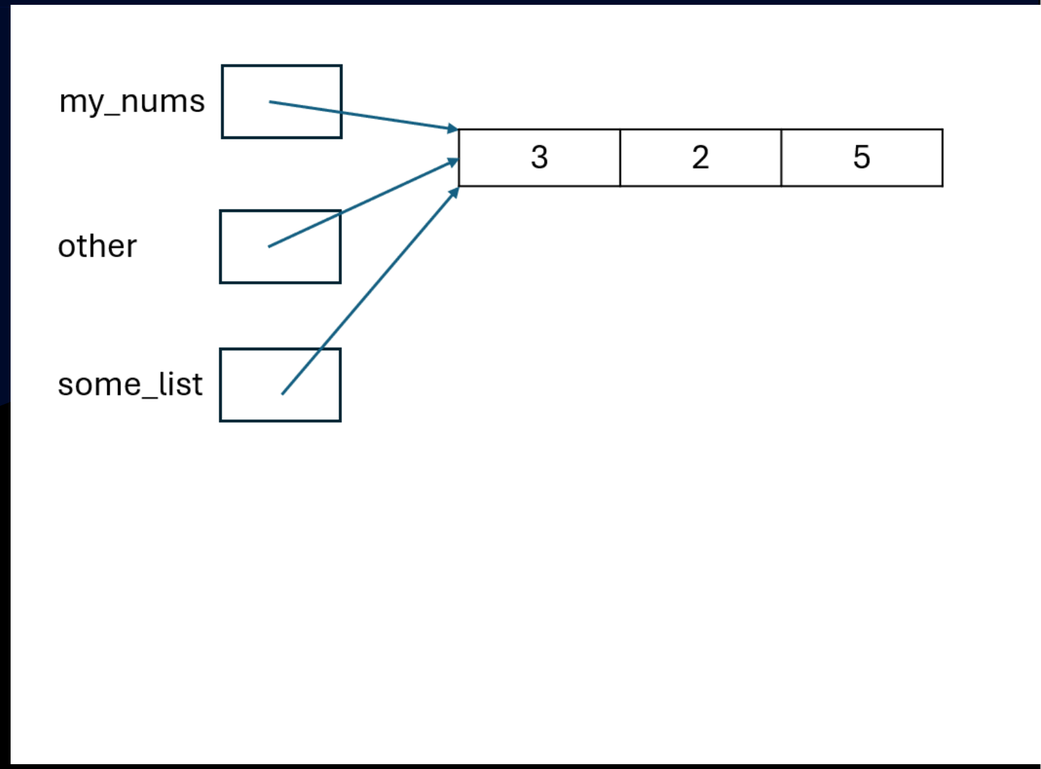
References to Mutable Types

References get more tricky when we start thinking about mutable types.

Consider:

```
def func(some_list):          # <-------
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)             # <-------
    other[1] = 1100
    print(my_nums)
```



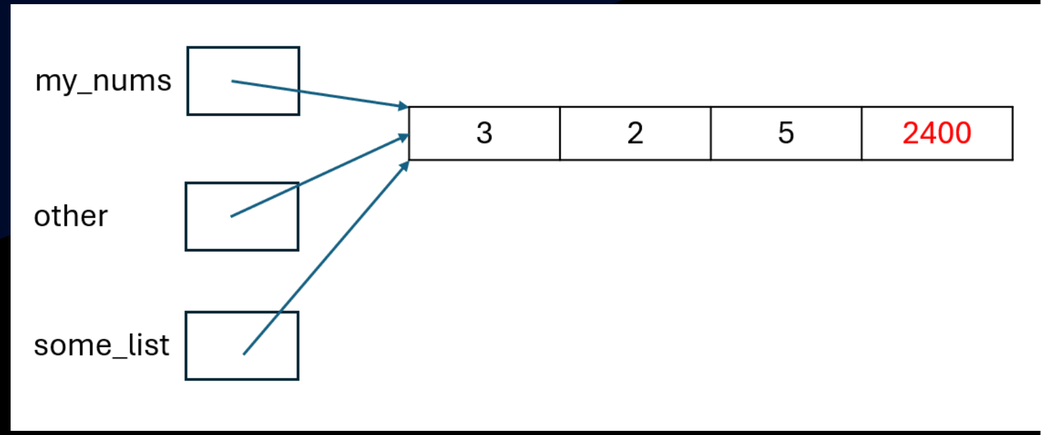
References to Mutable Types

References get more tricky when we start thinking about mutable types.

Consider:

```
def func(some_list):
    some_list.append(2400)  # <-------

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)
    other[1] = 1100
    print(my_nums)
```



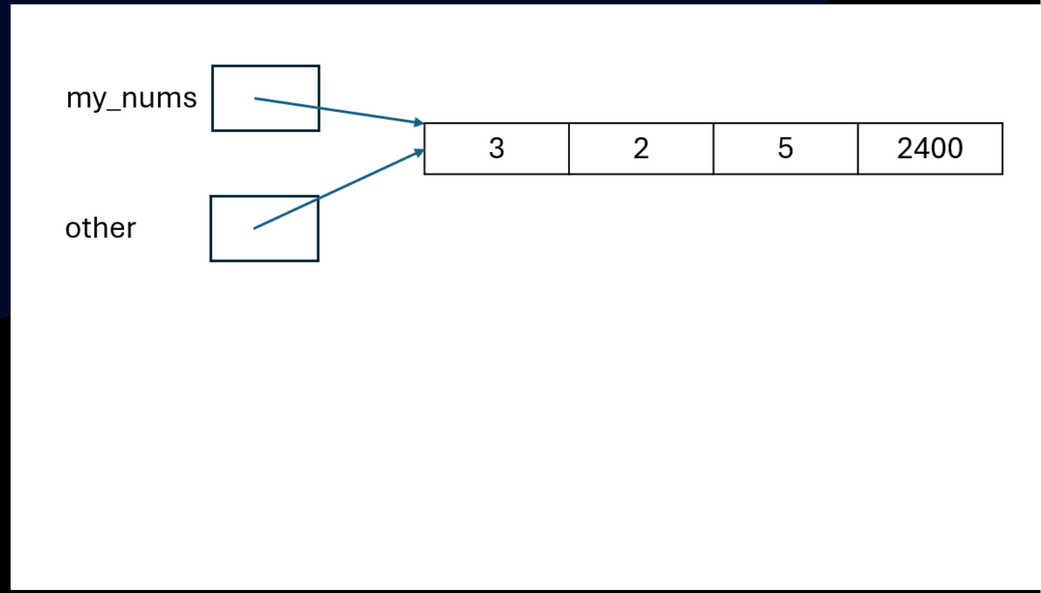
References to Mutable Types

References get more tricky when we start thinking about mutable types.

Consider:

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)              # <--------
    other[1] = 1100
    print(my_nums)
```



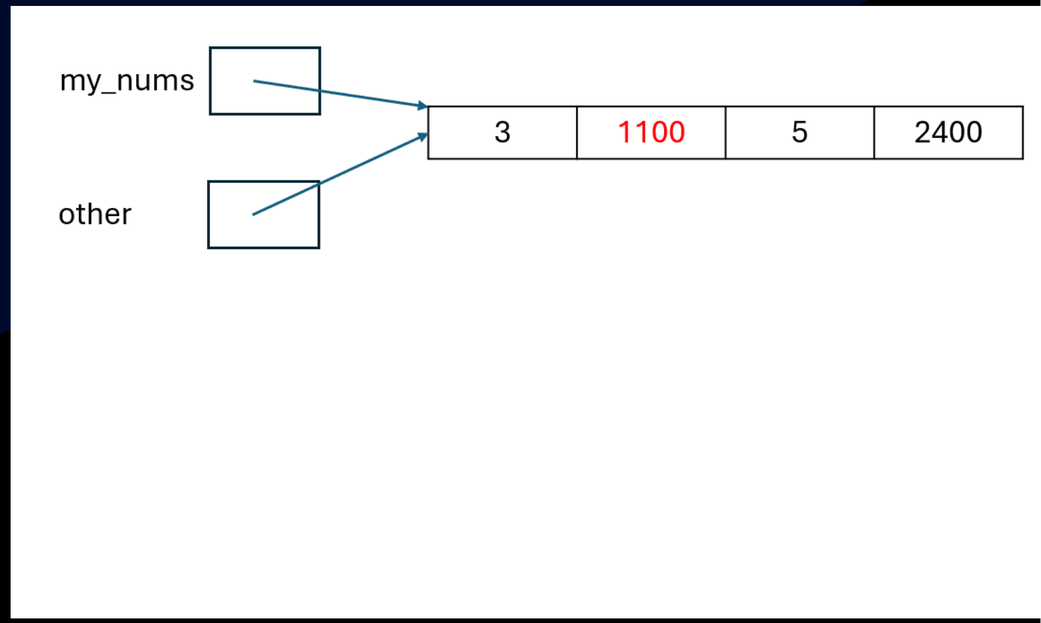
References to Mutable Types

References get more tricky when we start thinking about mutable types.

Consider:

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)
    other[1] = 1100          # <--------
    print(my_nums)
```



Practice

What gets printed?

S6

```
def add_five(num):
    num += 5

def main():
    x = 3
    add_five(x)
    print(x)
```

S7

```
def list_add_five(to_add):
    copy = to_add
    copy.append(copy[0] + 5)

def main():
    my_list = [3]
    list_add_five(my_list)
    print(my_list)
```

Practice

Given a class called `Point` with two fields, `x` and `y`, what gets printed?

(S10)

```
p = Point(x=2024, y=10)  # you can assume this works
not_p = p
not_p.x = 2015
p.x += 2

m = p.y
m += 1

print(p.x)
print(m)
print(p.y)
```

Review Data Class

If we wanted to make the `Point` object in the previous slide we would do:

```
from dataclasses import dataclass

@dataclass      # mark the class as a data class
class Point:    # Declare a class
    x: int      # declare the field names and their types
    y: int
```

In Python, a `dataclass` is the simplest kind of class.

- Defined (in most basic case) just by what properties that members of this class should have.

More advanced type annotations

If we want to have a data class with more advanced type notations, it would look something like this:

```
from dataclasses import dataclass

@dataclass
class Example:
    x: list[int]         # list of integers
    y: dict[str, int]    # dictionary, keys are strings, value are ints
    z: tuple[int, int, str]  # a tuple of two ints and a string
```

Practice:

(C12) Write a dataclass that represents a `Square` with three fields:

- a `float` to represent the `half_width`
- two more `float` values to represent the `center_x` and `center_y`
- a tuple containing three integers to represent the `color`

Next time

- More on objects and creating them!
- We will do some code that is VERY relevant for the next homework (FFF)